Principes de Programmation
Correction du Partiel 2017

6 juin 2017

Exercice 1 (Cours).

1. Quel est le type de `max` donné en annexe ?
   Sur 1,5 points.

   ```
   max :: (Ord a) => a -> a -> a
   ```

   La fonction `max` compare deux objets du même type `a` et renvoi le plus grand (toujours du type `a`). Cette fonction est polymorphe, ce qui veut dire que `a` peut être n'importe quoi. Sauf que l'on a besoin de comparer les deux entrées (ce qui n'est pas faisable pour certains types comme `(Int->Int)`), c'est pourquoi `a` doit être ordonné ce qui est indiqué par `(Ord a)` obligeant `a` à implémenter la type-class `Ord`.
   Remarque : le type d'une fonction correspond à sa signature, il indique à la fois le type des entrées et de la sortie.

2. Quel est le type de `map` donné en annexe ?
   Sur 1,25 points.

   ```
   map :: (a -> b) -> [a] -> [b]
   ```

   La fonction `map` prends deux argument :
   – une fonction `(f :: a -> b)` qui prend du `a` et renvoi du `b`,
   – une liste `(l :: [a])` qui contient des éléments de type `a`,
   elle renvoi alors une liste `(map f l :: [b])` d'éléments de type `b` correspondant à l'application de `f` sur chacun des éléments de `l`. Comme la fonction précédente (et comme la plupart des programmes fonctionnels), celle ci est polymorphe et les types `a` et `b` peuvent êtres n'importe lesquels.

3. Qu'est-ce qu'une *type-class* ? Donner un exemple.
   Sur 2 points.
   Une *type-class* est un concept propre à Haskell qui, moralement, sont aux types Haskell ce que les interfaces sont aux classes Java. Un type `a` implémente une *type-class* lorsque
   – il implémente une liste donnée de fonctions dont le type est paramétriques en `a`,

– il vérifie une liste de règle de "sanité" de l'implémention. (Ces règles devrait être vérifier aussi dans les interfaces Java, mais sont rarement aussi explicitement requises.)

Exemples vus en cours : `Eq`, `Ord`, `Show`, `Read`, `Num`, `Enum`, `Bounded`, [1] `Monoid`, `Functor`, `Monad`, `Foldable`...

4. Qu'est ce qu'une *monade*, moralement ?
   Sur 1,25 points.
   Une monade est moralement un environnement de travail. C'est à dire que l'on peut programmer normalement (sauf que les applications sont remplacés par des *binds*) tout en interagissant avec une environnement arbitrairement complexe.

5. Donnez trois nouveaux exemples (la définition formelle n'est pas requise).
   Sur 1,5 points.
   Exemples vus en cours : `Maybe` (gère une absence de résultats), `[]` (liste de *threads* primitifs), `IO` (gère l'interaction avec le système d'exploitation), `Exeption` [2] (gère des exceptions arbitrairement complexes), `State` [3] (gère une cellule mémoire globale), `Cont` (gère des "sauts" dans le code tels que des *breaks*) ; on a aussi vu `Yeld` (délègue la temporalité de l'exécution à son environnement) qui n'est pas un exemple de la bibliothèque standard.

Exercice 2 (Programmation).
Il y avait ici 2,25 points à récupérer sur l'utilisation correcte du patern matching et sur la propreté du code.
On définit les arbres de préfixes ainsi :

```
data PTree a = Noeud (Char -> PTree a)  |  Feuille a  |  Vide
```

Remarques :

– Le type `(Char -> PTree a)` est le type des fonctions qui transforment un caractère en un nouvel arbre préfixe. Donc `(Noeud f)` est un arbre préfixe qui pointe sur 128 237 nouveaux arbres préfixes (potentiellement différents), un pour chaque caractère. Par contre il n'y a aucune valeur dans un tels noeud, celles-ci ne sont présentes que sur certaines feuilles.

– Bien que leur nom le présent comme des arbres, les arbres préfixes sont plus proche des automates dont le type est

```
data Noeud    = Noeud Bool (Char -> Noeud a)
type Automate = Noeud
```

De fait, les programmes requis étaient plus proche des automates.

1. `rechercheParClef` qui cherche un élément dans l'arbre grâce à sa clef ; la clef d'un élément de l'arbre est le le String dont les lettres définissent le chemin d'accès.
   Sur 3 points.

1. Enum et Bounded valaient un peut moins de points car donnés en annexe.
2. ou Exept ou Error ou Failable qui sont des variantes
3. ou St qui est une variante

```haskell
rechercheParClef :: PTree a -> String -> Except String a
rechercheParClef (Noeud f  ) []    = throwError "La clef est trop courte"
rechercheParClef (Noeud f  ) (c:s) = rechercheParClef (f c) s
rechercheParClef (Feuille a) []    = return a
rechercheParClef (Feuille a) _     = throwError "La clef est trop longue"
rechercheParClef Vide        _     = throwError "La clef ne correspond a rien"
```

Remarquez que l'on a changé le type par rapport à celui demandé. Il était possible de le laisser tel quel en ne définissant pas les cas d'échec (en utilisant `undefined` ou en n'implémentant pas ces cas), il était aussi possible d'utiliser la monade `Maybe`.

2. `parser` qui utilise un arbre préfixes pour lire une phrase, rendant une liste des éléments (de types `a`) ; en effet, puisque les clefs sont des préfixes, une phrase ne peut être découpée qu'en une seule suite de clefs (plus éventuellement quelques lettres à la fin).
   Sur 2,5 points.
   Il faut ici utiliser une fonction auxiliaire

```haskell
parserAux :: PTree a -> String -> PTree a -> [a] -> Except (String,[a]) [a]
parserAux (Noeud f  ) ""    _ acc = throwError ("La phrase n'est pas terminee", acc)
parserAux (Noeud f  ) (c:s) t acc = parserAux (f c) s t acc
parserAux (Feuille a) ""    t acc = return (a:acc)
parserAux (Feuille a) (c:s) t acc = parserAux t     s t (a:acc)
parserAux Vide        _     _ acc = throwError ("La phrase ne correspond a rien",acc)

parser :: PTree a -> String  -> Except (String,[a]) [a]
parser t s = parserAux t s t []
```

   Ici encore, il était possible de ne pas recourir à `Except` en laissant des cas non définit, en utilisant `Maybe` ou simplement en rendant un liste interrompue (celle qui est ici rendue avec le message d'erreur).

3. `recherche` qui cherche si un élément est dans l'arbre,
   Sur 4 points.
   Il fallait ici utiliser les *type-class* `Enum` et `Bounded` (données en annexe) comme dans le TP6 exercice 3. En effet, elles permettent de lister tous les caractères du premier au dernier dans `[minBound, maxBound]`.

```haskell
recherche :: (Eq a) => a -> PTree a -> Bool
recherche a (Noeud f  )          = or (map ((recherche a).f) [minBound,maxBound])
recherche a (Feuille a') | a==a' = True
recherche _ t                    = False

or :: [Bool] -> Bool
or (b:l) = b || (or l)
```

4. `toList` qui donne la liste des éléments apparaissant l'arbre,
   Sur 3 points.

```
toList :: PTree a -> [a]
toList (Noeud f  ) = (toList.f) =<< [minBound,maxBound]
toList (Feuille a) = [a]
toList Vide        = []
```

5. Implémentez une *type-class* de votre choix
   Sur 2 points.
   Il était fortement préférable (mais pas obligatoire) d'implémenter la classe avec les arbres préfixe. On pouvait, par exemple, implémenter `Eq` :

```
instance (Eq a) => Eq (PTree a) where
  (Noeud f  ) == (Noeud g  ) = and (map (\c -> f c == g c ) [minBound,maxBound])
  (Feuille a) == (Feuille b) = (a == b)
  Vide        == Vide        = True
  t           == t'          = False
```

Exercice 3 (Programmation).
On considère l'implémentation suivante des AVL :

```
type Hauteur = Int
data AVL a = F | N Hauteur a (AVL a) (AVL a)

h F = 0
h (N n _ _ _) = n

left  (N _ _ f _) = f
right (N _ _ _ g) = g

n x f g = N (1 + max (h f) (h g))) f g

ajt F x = N 1 x F F
ajt (N k x f g) y
   | x==y = (N k y f g)
   | (x>y) && h (ajt f y) > (h g +1)  =
       (if (h (left (ajt f y)) >= h(right (ajt f y)) )
         then rotg else rotgd)
       (N (h (ajt f y)) x (ajt f y) g)
   | (x<y) && h (ajt g y) > (h f +1)  =
       (if (h (left (ajt g y)) <= h(right (ajt g y)) )
         then rotd else rotdg)
       (N (h (ajt g y)) x f (ajt g y))


rotg (N h x (N i y f gg) g) =  (N' y f (N' x gg g))
rotgd (N h x (N y f (N z ff gg)) g) = (N' z (N' y f ff) (N' x gg g))
rotd (N x g (N y ff g)) =  (N' y (N' x f ff) g)
rotdg (N x f (N y (N z ff gg) g)) = (N' z (N' x f ff) (N' y gg g))
```

Ce programme est en fait plein de défauts. Listez et corrigez les erreurs.

Remarque : Il s'agit du premier jet que j'ai écrit pour un code d'AVL, avant de faire les corrections et les amélioration requises. Vous pouvez ainsi voire que coder proprement, ce n'est pas coder proprement dès le premier jet, mais c'est bien être capable de nettoyer son code après coup.

Voici les défauts que vous pouviez trouver [4] (correction disponible dans un fichier `.hs` séparé) :

- Absence de typage. *(1 point si remarqué, 2 points si corrigé.)*
  Remarque : Donner le type n'est généralement pas nécessaire à la correction, mais il orienté énormément le lecteur dans sa compréhension du code et permet de repérer certains bugs.
- Absence de commentaires. *(1 point si remarqué, 2 points si corrigé.)*
- Mauvais noms de fonctions/variables/constructeurs. *(1 point si remarqué, 2 points si corrigé.)*
- Parenthèse fermente en trop en ligne 10 *(0.25 points)*.
- Argument manquant pour `N` en ligne 10 *(1 point si remarqué)*. Il s'agit de l'argument `x` qui autrement n'était pas utilisé *(1 point de correction)*.
- Argument de hauteur oublié dans le *patern-matching* des lignes 27 et 28 *(0.5 point si remarqué)*.
- Constructeur `N'` des lignes 25 à 28 non définit *(1 point si remarqué)*. Il s'agissait en fait de la fonction `n` définit ligne 10 *(1 point de correction)*.
- Les *patern-machings* (de `h`, `left`, `right`, `rotg`, `rotgd`, `rotd` et `rotdg`) ne sont pas complets, le cas des feuilles est souvent absent *(0.5 point si remarqué)*. Il s'agit de cas où l'on n'est pas sensé arriver, il faut donc y relever des erreurs *(1 point de correction)*.
  Remarque : Le programme va normalement tourner malgré ça, mais c'est de la mauvaise programmation car il sera difficile de débugger et le lecteur aura plus facilement un doute sur la correction de code (il est facile d'oublier réellement un cas).
- Les case des gardes de `ajt` n'est pas complet : il manque les cas où il n'y a pas de rotations *(1 point si remarqué et 2 points de correction)*.
- Les noms des rotations gauche et droites sont inversés, mais pas leur utilisation [5] *(1,5 point)*.
- Aux lignes 15, 16 et 17, le même calcul de `(ajt f y)` est fait 4 fois, ce qui explose la complexité, idem sur la troisième garde *(1 point si remarqué)*. Il faut donc le mettre ce calcule dans un `where` histoire de ne le faire qu'une fois *(1 point de correction)*.
- Il aurait été mieux d'écrire `rotgd` et `rotdg` à partir de `rotg` et `rotd` *(0,5 point)*.
- En ligne 15 (et ligne 19), l'expression `(h g 1)+` calcule bien la hauteur de `g` avant d'ajouter `1`, mais prête à confusion (si on ne connaît pas bien les priorités). Il vaut mieux marquer `(h g) 1+`. *(0.25 points en correction.)*

4. Il était impossible de tout trouver et corriger en 2 heures à moins de faire que ça... ça tombe bien il y avait 20 points à récupérer sur cet exo...

5. Le programme fonctionne bien, j'ai juste du mal avec ma gauche et ma droite lorsque l'on parle de rotation...

– En lignes 16, 17 et 18 (idem en lignes 20, 21, 22), on choisi la rotation avec un if dans une gosse parenthèse puis on lui applique son argument. C'est correcte mais très sale. Il vaut mieux utiliser une fonction annexe. *(0,5 point si remarqué + 2 si corrigé.)*
– Il n'y a aucune référence à l'algo utilisé *0.5 points.*
– On aurait peu rajouter de la modularité comme des classes dérivées, un module au dessus, etc... *(Jusque 2 points si implémentés dans la corrections)*

Exercice 4 (Egalités). (Questions 1,2,3,4 non nécessaires pour 5,6,7,8)
On définit la classe des monades mathématiques comme un foncteur spécialisé :

```
class Functor m => MathMonad m where
   unit :: a -> m a
   mult :: m (m a) -> m a
```

En plus des règles du foncteur, une monade mathématique doit respecter les règles suivantes (pour tout `x::a` `l::m a`, `u: m(m a)`, `v::m(m(m a))` et `f::a->b`) :

```
mult (unit l)      = l                       :: m a
mult (fmap unit l) = l                       :: m a
mult (mult v)      = mult (fmap mult v)      :: m a
fmap f (unit x)    = unit (f x)              :: m b
fmap f (mult u)    = mult (fmap (fmap f) u)  :: m b
```

Errata : Les deux dernières équations ont été oubliées dans l'énoncer original. Elles rendent plus complexe la question 4 et plus simple la question 6...

(2 points de disponible sur la compréhension et l'utilisation du concept d'égalité de programme.)

1. Rappelez les équations que doivent satisfaire un foncteur (et donc une monade mathématiques).
   Un foncteur `funct` doit satisfaire les équations suivantes pour toutes fonctions `(f::b->c)` et `(g::a->b)` et tout `(x:: funct a)` :

   ```
   fmap id x     =   x                    :: funct a
   fmap (f.g) x  =   fmap f (fmap g x)    :: funct c
   ```

   (1 point)
   Remarque : la quantification et le typage n'étaient pas requis.
2. Rappelez la définition de la classe `Monad`.
   C'est la définition formelle *(1 point)* :

   ```
   class Monad m where
     return :: a -> m a
     (=<<)  :: (a-> m b) -> m a -> m b
   ```

accompagnée des égalités requises. Ainsi pour tout `(x :: m a)`, `(y :: b)`, `(f :: b -> m c)` et `(g :: a -> m b)`, on doit avoir *(1 point)* :

```
return =<< x       =   x                          :: m a
f =<< (return y)   =   f y                        :: m c
f =<< (g =<< x)    =   (\y. f =<< (g y) ) =<< x   :: m c
```

3. Montrez que toute monade est une monade mathématique.

   Remarque : Cette question et la suivante sont de loin les deux plus complexes de l'examen. Je ne n'attendais pas à ce que vous les résolviez, mais je voulais voire comment la question était abordée.

   Soit `m` une monade.
   On sait que `m` a deux fonctions de définies `return`, `(=<<)`. On sait de par le cours (et le TD 3) que toute monade est un foncteur avec :

```
instance Functor m where
  fmap f x = (return.f) =<< x
```

   Il faut maintenant en faire une mathMonad. Pour ça on définit d'abord les fonctions :

```
instance MathMonad m where
   unit   = return
   mult l = id =<< l
```

   On vérifie que `return` a bien le même type que `unit` (trivial). On vérifie de plus que lorsque `(l :: m (m a))` on a bien `(id =<< l :: m a )` car `(id :: m a -> m a)` (en particulier), ce qui est le type recherché de `(mult l)`.
   On doit aussi vérifier aussi que les 5 égalités de la monade mathématique sont respectées. Ainsi, pour tout `x::a` `l::m a`, `u: m (m a)` et `v::m(m(m a))` :

```
mult (unit l)      = id =<< (unit l)                -- def de mult
                   = id =<< (return l)              -- def de unit
                   = id l                           -- equation 2 de Monade
                   = l                              -- def de id
mult (fmap unit l) = id =<< (fmap unit l)           -- def de mult
                   = id =<< (fmap return l)         -- def de unit
                   = id =<< ((return.return) =<< l) -- def de fmap
                   = (\y. id =<< ((return.return) y) ) =<< x
                                                    -- equation 3 de Monade
                   = (\y. id =<< (return (return y)) ) =<< x
                                                    -- def composition
                   = (\y. id (return y)) =<< x      -- equation 2 de Monade
                   = (\y. return y) =<< x           -- def de id
                   = return =<< x                   -- return est une fonction
                   = x                              -- equation 1 de Monade
mult (mult v)      = id =<< (id =<< v)              -- def de mult
```

```
                    = (\y. id =<< (id y) ) =<< x      -- equation 3 de Monade
                    = (\y. id =<< y ) =<< x           -- def de id
                    = (\y. mult y) =<< x              -- def de mult
                    = (\y. id (mult y)) =<< x         -- def de id
                    = (\y. id =<< (return (mult y))) =<< x
                                                      -- equation 2 de Monade
                    = (\y. id =<< ((return.mult) y)) =<< x
                                                      -- def composition
                    = id =<< ((return.mult) =<< v)    -- equation 3 de Monade
                    = id =<< (fmap mult v)            -- def de fmap
                    = mult (fmap mult v)              -- def de mult
fmap f (unit x)     = (return.f) =<< (unit x)         -- def de fmap
                    = (return.f) =<< (return x)       -- def de unit
                    = (return.f) x                    -- equation 2 de Monade
                    = return (f x)                    -- def composition
                    = unit (f x)                      -- def de unit
fmap f (mult u)     = (return.f) =<< (mult u)         -- def de fmap
                    = (return.f) =<< (id =<< u)       -- def de mult
                    = (\y. (return.f) =<< (id y) ) =<< x   -- equation 3 de Monade
                    = (\y. (return.f) =<< y) =<< x    -- def de id
                    = (\y. fmap f y) =<< x            -- def de fmap
                    = (\y. id (fmap f y)) =<< x       -- def de id
                    = (\y. id =<< (return (fmap f y)) =<< x
                                                      -- equation 2 de Monade
                    = (\y. id =<< ((return.(fmap f)) y)) =<< x
                                                      -- def composition
                    = id =<< ((return.(fmap f)) =<< u)
                                                      -- equation 3 de Monade
                    = id =<< (fmap (fmap f) u)        -- def de fmap
                    = mult (fmap (fmap f) u)          -- def de mult
```

4. Inversement, montrez que toute monade mathématique est une monade.

   Remarque : Cette question et la précédente sont de loin les deux plus complexes de l'examen. Je ne n'attendais pas à ce que vous les résolviez, mais je voulais voire comment la question était abordée.

   Soit `m` une monade mathématique.
   On sait que `m` a trois fonctions de définies `unit`, `mult` et `fmap`. Il faut définir `return` et `(=<<)` pour en faire une monade :

   ```
   instance Monad m where
     return  = unit
     f =<< x = mult (map f x)
   ```

   On vérifie que `unit` a bien le même type que `return` (trivial). On vérifie de plus que lorsque `(f :: a -> mb)` et `(x::a)` on a bien `(map f x :: m (m b) )` de façon à ce que `(mult (map f x) :: m b )` qui est le type recherché de `(f =<< x)`.

On doit aussi vérifier aussi que les 3 égalités de la monade sont respectées. Ainsi pour tout `(x :: m a)`, `(y :: b)`, `(f :: b -> m c)` et `(g :: a -> m b)`, on a :

```
return =<< x      = mult (fmap return x)        -- def du bind de m
                  = mult (fmap unit   x)        -- def du return de m
                  = x                           -- equation 3 de MathMonad
f =<< (return y) = mult (fmap f (return y))     -- def du bind de m
                  = mult (fmap f (unit   y))    -- def du return de m
                  = mult (unit (f y)      )     -- equation 4 de MathMonad
                  = f y                         -- equation 1 de MathMonad
f =<< (g =<< x)  = mult (fmap f ( mult (fmap g x)))
                                                -- def du bind de m
                  = mult (mult (fmap (fmap f) (fmap g x)))
                                                -- equation 5 de MathMonad
                  = mult (fmap mult (fmap (fmap f) (fmap g x)))
                                                -- equation 3 de MathMonad
                  = mult (fmap (mult.(fmap f).g) x)
                                                -- equation 2 du Foncteur
                  = mult (fmap (\y. mult (fmap f (g y))) x)
                                                -- def de la composition
                  = (\y. f =<< (g y) ) =<< x
                                                -- def du bind de m
```

Comme dans l'exercice 2, on définit les arbres de préfixe ainsi :

```
data PTree a = Noeud (Char -> PTree a)  |  Feuille a |Vide
```

5. Donnez une implémentation de `Functor PTree`.
   (1 point)

   ```
   instance Functor PTree where
     fmap f (Noeud g  ) = Noeud (\c -> fmap f (g c))
     fmap f (Feuille a) = Feuille (f a)
     fmap _ Vide        = Vide
   ```

6. Montrez que les arbres de préfixe forment bien un foncteur.
   (2 point)
   Pour toutes fonctions `(f1::b->c)` et `(f2::a->b)` et tout `(t:: PTree a)`, nous procédons par récurrence sur `t` de sorte à avoir trois cas : `(t=Noeud g  )`, `(t=Feuille y)` et `(t=Vide)`

   ```
   fmap id (Noeud g  )      = Noeud (\c -> fmap id (g c)) -- def de fmap
                            = Noeud (\c -> g c)           -- hypothese de recurence
                            = Noeud g                     -- g est une fonction
   fmap id (Feuille y)      = Feuille (id y)              -- def de fmap
                            = Feuille y                   -- def de id
   fmap id Vide             = Vide                        -- def de fmap
   fmap (f1.f2) (Noeud g  ) = Noeud (\c -> fmap (f1.f2) (g c))
   ```

```
                                        -- def de fmap
= Noeud (\c -> fmap f1 (fmap f2 (g c)))
                                        -- hypothese de recurence
= Noeud (\c -> fmap f1 ((\c'-> fmap f2 (g c')) c))
                                        -- beta equivalence
= fmap f1 Noeud (\c'-> fmap f2 (g c'))
                                        -- def de fmap
= fmap f1 (fmap f2 (Noeud g)) -- def de fmap
```

7. Donnez une implémentation de `MathMonad PTree`.
   (1 point)

```
instance MathMonad PTree where
   unit x           = Feuille x
   mult (Noeud f)   = Noeud (\c -> mult (f c))
   mult (Feuille t) = t
   mult Vide        = Vide
```

8. Montrez que les arbres de préfixe forment bien une monade. *(2 point)*

   On va montré qu'il s'agit bien d'une monade mathématique et conclure par la question 4. On utilise les question 5 et 6 pour prouver que c'est un foncteur. On a alors à prouver les équation de monade mathématique :
   Soit `t::Ptree a` :

```
mult (unit t) = mult (Feuille t)    -- def de unit
              = t                   -- def de mult
```

   Soit `t::Ptree a`, on fonctionne par récurrence avec les cas `(t=Noeud g)`, `(t=Feuille y)` et `(t=Vide)` :

```
mult (fmap unit (Noeud g)  ) = mult (Noeud (\c -> fmap unit (g c)))
                                                    -- def de fmap
                             = Noeud (\c' -> mult ((\c -> fmap unit (g c)) c'))
                                                    -- def de mult
                             = Noeud (\c' -> mult (fmap unit (g c')))
                                                    -- beta equivalence
                             = Noeud (\c' -> g c')  -- hypothese de recurrence
mult (fmap unit (Feuille y)) = mult (Feuille (unit y))  -- def de fmap
                             = unit y               -- def de mult
                             = Feuille y            -- def de unit
mult (fmap unit Vide       ) = mult Vide            -- def de fmap
                             = Vide                 -- def de mult
```

   Soit `v:: Ptree (Ptree a)`, on fonctionne par récurrence sur `v` avec les cas `(v=Noeud g)`, `(v=Feuille t)` et `(v=Vide)` :

```
mult (mult (Noeud g)  ) = mult (Noeud (\c -> mult (g c)))
                                                    -- def de mult
```

```
                          = Noeud (\c' -> mult ((\c -> mult (g c)) c'))
                                                          -- def de mult
                          = Noeud (\c' -> mult (mult (g c')))
                                                          -- beta equivalence
                          = Noeud (\c' -> mult (fmap mult (g c')))
                                                          -- hypothese de recurrence
                          = Noeud (\c' -> mult ((\c -> fmap mult (g c)) c'))
                                                          -- beta equivalence
                          = mult (Noeud (\c -> fmap mult (g c)))
                                                          -- def de mult
                          = mult (fmap mult (Noeud g c))  -- def de fmap
mult (mult (Feuille t)) = mult t                          -- def de mult
                          = mult (Feuille (mult t))       -- def de mult
                          = mult (fmap mult (Feuille t))  -- def de fmap
mult (mult Vide)        = mult Vide                       -- def de mult
                          = mult (fmap mult Vide)         -- def de fmap
```

Soit `x::a` et `f :: a->b` :

```
fmap f (unit x) = fmap f (Feuille x)          -- def de unit
                = Feuille (f x)               -- def de fmap
                = unit (f x)                  -- def de unit
```

Soit `(u :: Ptree (Ptree a))` et `(f :: a->b)`. Par par récurrence sur `u` avec les cas `(u=Noeud g)`, `(u=Feuille t)` et `(u=Vide)` :

```
fmap f (mult (Noeud g  )) = fmap f (Noeud (\c -> mult (g c)))
                                                          -- def de mult
                          = Noeud (\c' -> fmap f ((\c -> mult (g c)) c'))
                                                          -- def de fmap
                          = Noeud (\c' -> fmap f (mult (g c')))
                                                          -- beta equivalence
                          = Noeud (\c' -> mult (fmap (fmap f) (g c')))
                                                          -- hypothese de recurrence
                          = Noeud (\c' -> mult ((\c->(fmap (fmap f) (g c))) c'))
                                                          -- beta equivalence
                          = mult (Noeud (\c->(fmap (fmap f) (g c))))
                                                          -- def de mult
                          = mult (fmap (fmap f) (Noeud g))
                                                          -- def de fmap
fmap f (mult (Feuille t)) = fmap f t                      -- def de mult
                          = mult (Feuille (fmap f t))     -- def de mult
                          = mult (fmap (fmap f) (Feuille t))
                                                          -- def de fmap
fmap f (mult Vide       ) = fmap f Vide                   -- def de mult
                          = Vide                          -- def de fmap
                          = mult Vide                     -- def de mult
                          = mult (fmap (fmap f) Vide)     -- def de fmap
```